Efficient Operations On MDDs For Building Constraint Programming Models

Guillaume Perez, Jean-Charles Régin

Efficient Operations On MDDs For Building Constraint Programming Models

Guillaume Perez and Jean-Charles Régin
Université Nice-Sophia Antipolis, I3S UMR 7271, CNRS, France
guillaume.perez06@gmail.com; jcregin@gmail.com

Abstract

We propose improved algorithms for defining the most common operations on Multi-Valued Decision Diagrams (MDDs): creation, reduction, complement, intersection, union, difference, symmetric difference, complement of union and complement of intersection. Then, we show that with these algorithms and thanks to the recent development of an efficient algorithm establishing arc consistency for MDD based constraints (MDD4R), we can simply solve some problems by modeling them as a set of operations between MDDs. We apply our approach to the regular constraint and obtain competitive results with dedicated algorithms. We also experiment our technique on a large scale problem: the phrase generation problem and we show that our approach gives equivalent results to those of a specific algorithm computing a complex automaton.

1 Introduction

MDDs become more and more used in constraint programming. The recent development of MDD4R, an efficient and fully incremental arc consistency algorithm [Perez and Régin, 2014], seems to show that MDD constraints are competitive with Table constraints. We can see a table constraint as a set of disjoint paths (each path corresponding to a tuple) that are preceded by an initial node and whose last node is linked to a terminal node, that is as a particular MDD which is not compressed. The compression process may dramatically reduce the size of the underlying graph and so the complexity of the arc consistency algorithm maintaining it.

In this paper, we aim at showing that we can expect to directly use MDDs with the help of MDD4R for establishing arc consistency of some complex constraints like the regular constraint or constraints based on dynamic programming like the knapsack constraint. We propose to explicitly create and work with MDDs instead of dedicated algorithms associated with the constraints. The advantage of dealing directly with MDDs is that they are always reduced after their creation so MDD4R may outperform dedicated algorithms, which do not perform this operation. Currently, the complexity of the existing algorithms prevents us from using them for solving some problems. For instance, phrase generation problem involves domains having more than $d = 10,000$ values. Thus, we cannot use an algorithm whose time or space complexity is mainly based on $\Omega(nd)$, where n is the number of nodes of the MDDs. Therefore, we need to improve the algorithms performing the main operations on MDDs: creation, reduction and combinations.

The new creation algorithm we propose, exploits the origin of the definition of the MDD. If the MDD represents an automaton (like with a regular constraint) or a repeated pattern (like with dynamic programming), then its creation may be sped-up.

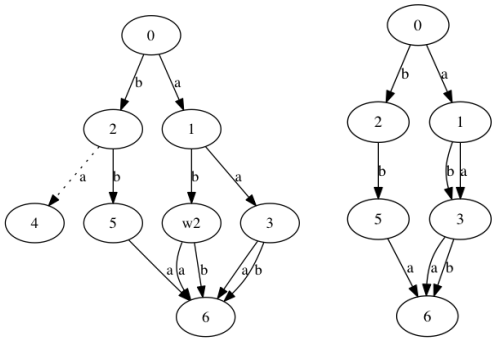
Then, we introduce pREDUCE a new algorithm for reducing an MDD, i.e. eliminating nodes that are not lying on a path from the initial node to the terminal node. Our algorithm proceeds by layers and uses a simple an original procedure for gathering equivalent nodes in order to keep the time complexity in the size of the neighborhood of nodes instead of being bounded by $\Omega(d)$. This new algorithm improves the Cheng and Yap's one [Cheng and Yap, 2010]. It is also conceptually simpler.

At last, we present a generic algorithm for computing basic combinations of MDDs. Our algorithm is based on graph operations that are made on the underlined graph of the MDDs. Each combination, such as intersection, union, difference, symmetric difference, is defined by applying a specific operation for each node of the MDDs depending on the fact that two nodes have an arc labeled by a value or not. In that way, we obtain a really short and powerful algorithm. Then by defining some operations for each node of each layer we instantiate our generic algorithm and compute the desired combination.

Before concluding, we present some experiments showing some strong improvements brought by our algorithms. In particular, we show that we can gain orders of magnitude for the reduction of MDDs and that we are able to compute a dozen of intersections of MDDs having millions of nodes, hundreds millions of arcs and ten thousand values. Then, the application of MDD4R on MDDs leads to computational times equivalent to those of dedicated algorithms.

2 Background

Multi-valued decision diagram (MDD) is a method for representing discrete functions. It is a multiple-valued exten-



Mdd before reduction Mdd after reduction

Figure 1: Reduction of an MDD. Nodes without successors are deleted (e.g. node 4). Equivalent nodes are merged (e.g. nodes 3 and w2).

sion of BDDs [Bryant, 1986]. An MDD, as used in CP [Andersen *et al.*, 2007; Hadzic *et al.*, 2008; Hoda *et al.*, 2010; Bergman *et al.*, 2011; Gange *et al.*, 2011], is a rooted directed acyclic graph (DAG) used to represent some multivalued function $f : \{0...d-1\}^r \rightarrow \{true, false\}$, based on a given integer d (See Figure 1.). Given the r input variables, the DAG representation is designed to contain r layers of nodes, such that each variable is represented at a specific layer of the graph. Each node on a given layer has at most d outgoing arcs to nodes in the next layer of the graph. Each arc is labeled by its corresponding integer. The final layer is represented by the true terminal node (the false terminal node is typically omitted). There is an equivalence between $f(v_1, ..., v_r) = true$ and the existence of a path from the root node to the true terminal node whose arcs are labeled $v_1, ..., v_r$. Nodes without any outgoing arc or without any incoming arc are removed.

In an MDD constraint, the MDD models the set of tuples satisfying the constraint, such that every path from the root to the true terminal node corresponds to an allowed tuple. Each variable of the MDD corresponds to a variable of the constraint. An arc associated with an MDD variable corresponds to a value of the corresponding variable of the constraint.

We will denote by $L[i]$ the nodes in layer i and by $\omega^+(x)$ the set of outgoing arcs of the node x.

3 Creation

Some constraints like table, regular or knapsack may be explicitly represented by an MDD. The advantage of this approach is that the MDD is reduced after its creation and arc consistency algorithms may benefit from its reduction. The drawbacks are the time for creating the MDD and the complexity of the filtering algorithm. We will present later some experiments showing that MDD4R is competitive with dedicated algorithms. Now, we propose to show how the creation may be accelerated.

Cheng and Yap [Cheng and Yap, 2010] have proposed an algorithm for creating an MDD from a table contraint. In this case, a trie data structure is created and it is reduced in order to obtain an MDD. We introduce two new ways for creating an MDD: from an automaton and from a pattern.

Creation from an automaton. The regular constraint [Pesant, 2004] deals with an automaton from which it deduces the allowed tuples. It corresponds to a set of ternary transition constraints [Beldiceanu *et al.*, 2004]: $T(\delta, Q_i, x_i, Q_{i+1})$, where δ is a common transition function, Q_i and Q_{i+1} two state variables and x_i a variable. A transition constraint defines for which values of x_i we can go from a state q to a state q'. Note that δ is a function, so there is only one value of q' for a couple (v, q) with $v \in x_i$ and $q \in Q_i$. First, we create the nodes. Each layer i is associated with the variable Q_i and has as many nodes as there are possible states. Each node of the layer i corresponds to a pair (Q_i, q) where q is a possible state. Thus, for each tuple (q_1, v, q_2) satisfying the transition constraint we create for each layer i, an arc from node (Q_i, q_1) to node (Q_{i+1}, q_2) labeled by v. If $i = r$ then all nodes (Q_{i+1}, q_2) represent the true terminal node. Note that we do not need to know all tuples before creating the MDD.

It is important to remark that each layer of the MDD cannot have more arcs than the number of tuples in the transition constraint. So, using explicitly an MDD does not introduce any additional cost if the arc consistency algorithms of the ternary constraints do not share a global transition table.

Creation from a pattern. In some cases, the transition constraint depends on the variables it involves and Function δ is not globally shared. Precisely we have a δ_i function for each layer i. The layers are built successively by applying the transition constraint. Each node of the MDD is associated with an information that depends on the layer. For instance for the knapsack constraint [Trick, 2003], the information corresponds to the current value of the sum from the root to the current node (every path to the node leads to the same value). This information can be a scalar but also be more complex. Note that we may have an exponential number of nodes. However, only one layer in memory is sufficient for building the MDD.

4 Reduction

The reduction of an MDD is one of the most important operations. It consists of merging equivalent nodes, i.e. nodes having the same set of outgoing neighbors associated with the same labels. Figure 1 gives an example of reduction.

A reduction algorithm for BDDs has been given by Bryant [Bryant, 1986]. It proceeds by layer from the bottom to the top, considers each node x successively and tries to find another node x' that can be merged with it. It has a time complexity in $O(n \log(n))$ for a BDD having n nodes. We could adapt it for MDDs by associating with each node its list of neighbors. Note that the neighbor of a node can be seen as a couple $(node, label)$. Then, by ordering the neighbors lists, we can maintain an ordered list of nodes. At the beginning the list of nodes contains all nodes and then we reduce it by merging nodes having the same list of neighbors. If we have n nodes then the complexity of this algorithm will be in $O(n \log(n)d)$ because the comparison is in $O(d)$ for an MDD.

Cheng and Yap [Cheng and Yap, 2010] have proposed another algorithm (`mddify`) that also considers the nodes suc-

cessively and searches for merging nodes. However, it proceeds differently than Bryant's algorithm in order to improve the complexity. It uses a dictionary for speeding up the search for a similar node of a given node. Chang and Yap do not detail their algorithm and claim that the search for a similar node can be computed in $O(d)$. There are two ways for implementing such a dictionary that deserves some attention: by a radix tree or by an hash table.

First, consider a radix tree and that l is the length of a string and k the size of the alphabet. If we want an $O(l)$ complexity then we need to be able to traverse the tree from the root to a leaf in $O(l)$. Since the tree may have a depth equal to l it means that we need a random access to any child of a node. This can be achieved by associating an array of size k with each node of the radix tree. Thus, the space complexity for such a tree having p nodes is in $O(pk)$. It is not straightforward to use a radix tree for our problem because the neighbor of a node is defined by a couple (v, y) where v is the label (i.e. the value) and y the other extremity of the arc. For instance, the arcs (x_1, v, y_1) and (x_2, w, y_1) prevent the nodes x_1 and x_2 from being merged. Similarly, we have the same result for the arcs (x_1, v, y_1) and (x_2, v, y_2). A workaround consists of associating an array of d values with each node of the MDD. An entry v of this array corresponds to the node that can be reached with an arc labeled by v, or to nil if there is none. This array defines the string of a node. The number of possible letters becomes the number of nodes of the considered layer. Thus, by using a radix tree we can search for a similar node in $O(d)$ with an $O(pr)$ space complexity, if the radix tree has p nodes. Unfortunately, this can prevent us from using this algorithm because the search is in $\Omega(d)$ and not in the number of neighbors.

The second possibility is to use an hash table. In this case we cannot ensure to reach an $O(d)$ time complexity but we can expect the search in the table to be close to O(1) once the hash code of the key has been computed, which is in $O(d)$. Such a result can be obtained by using a table whose size is greater than n when n elements are involved. The drawback of this approach in practice is that it may be time consuming to compute an efficient hash code or we need a large table.

4.1 pReduce Algorithm

We propose PREDUCE, a new algorithm whose time complexity per node is bounded by its number of neighbors and not by the number of values per domain. In addition, it does not increase the memory consumption of the MDD. Instead of checking for each node whether it can be merged or not, we propose to consider all nodes of a layer as a whole and to build a general data structure. Then, we will compute the nodes of that layer that can be merged together. We assume that we want to merge the nodes of given layer i. Let $L[i]$ be this node set. The main idea is to consider the neighbors of nodes in $L[i]$ by their order of appearance in the neighborhood list. We will denote by $neigh[x][k]$ the k^{th} neighbor of node x, that is a couple (v, y) where v is a value and y a node. We will also denote by $node(c)$ (resp. $value(c)$), the node (resp. the value) of the couple c. Then, we work by considering the positions in the neighborhood list and build the sets of nodes having a common prefix (i.e. ordered neighbors) and we split them until a set becomes a singleton or all the neighbors have been considered.

Algorithm 1 pReduce of an MDD.

```
PREDUCE(L)
    define V_A, N_A, V_list, N_list
    for each i from r − 1 to 0 do
        REDUCELAYER(L[i], V_A, N_A, V_list, N_list)

REDUCELAYER(Layer, V_A, N_A, V_list, N_list)
    delete nodes without outgoing neighbors
    define the pack p with Layer, ∅, 0
    Q ← ∅
    REDUCEPACK(p, V_A, N_A, V_list, N_list, Q)
    while Q ≠ ∅ do
        pick and remove p from Q
        REDUCEPACK(p, V_A, N_A, V_list, N_list, Q)

REDUCEPACK(p, V_A, N_A, V_list, N_list, Q)
    i ← pos(p)
    for each x ∈ p do
        v ← value(neigh[x][i + 1])
        if V_A[v] = ∅ then add v to V_list
        add x to V_A[v]
    for each v ∈ V_list do
        for each x ∈ V_A[v] do
            y ← node(neigh[x][i + 1])
            if N_A[y] = ∅ then add (v, y) to N_list
            add x to N_A[y]
        V_A[v] ← ∅
        for each (v, y) ∈ N_list do
            if |N_A[y]| > 1 then
                define a pack p' with ∅, (v, y), i + 1
                M ← {x ∈ N_A[y]/|N(x)| = i + 1}
                merge all elements of M together
                add N_A[y] − M to p'; add p' to Q
            N_A[y] ← ∅
        N_list ← ∅
    V_list ← ∅
```

We introduce the notion of pack. Intuitively, a pack contains all the nodes having the same k first neighbors (i.e. prefix). Formally, a pack p is the set of nodes of the layer l associated with a position, denoted by $\texttt{pos}(p)$ and a set of $\texttt{pos}(p)$ couples (node,value), denoted by $\texttt{couple}(p, i)$ with $i = 1..\texttt{pos}(p)$ such that
$x \in p \Leftrightarrow \forall i = 1..\texttt{pos}(p) : neigh[x][i] = \texttt{couple}(p, i)$.

Then, we immediately have:

Property 1 *Given $x_1, x_2 \in L$ with $|N(x_1)| = |N(x_2)|$. Then, x_1 and x_2 can be merged if and only if $\exists$ pack p with $x_1 \in p, x_2 \in p$ and $\texttt{pos}(p) = |N(x_1)|$*

We can also note that if a pack contains only one node then this node cannot be merged with any node. We will denote by $|p|$ the number of nodes contained in p.

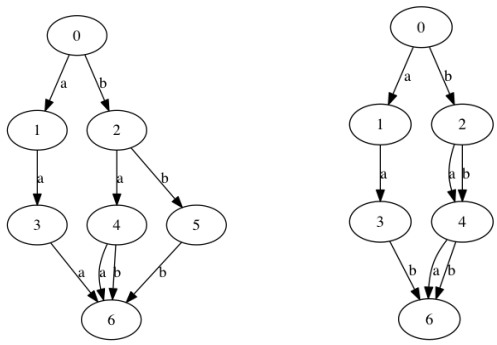


Figure 2: MDD operands. $\overline{mdd_1 \cap mdd_2}$ is given in Fig 1.

The computation of packs can be done by splitting other packs. A pack p can be divided into a set $S(p)$ of disjoint packs. Let i be the position of p, we will build a new pack for each different value of $neigh[x][i+1]$ with $x \in p$ and we will add a node $y \in p$ to the pack p' with $neigh[y][i+1] = \texttt{couple}(p', i+1)$.

Algorithm 1 is a possible implementation of this operation. It uses V_A and N_A two arrays of sets in order to perform the split operation on a pack p in $O(|p|)$. Array V_A is indexed by the values and array N_A is indexed by the nodes. Note that these two arrays contain only empty sets at the beginning and at the end of the operation. It also uses V_{list} and N_{list} two lists of elements that are used to save the entries that are not empty in the arrays. At the end of the algorithm the lists are empty. Algorithm 1 has two phases. First, it splits the current pack p into the array of sets V_A according to the value labeling the neighbor at position $\texttt{pos}(p)+1$. The second phase considers each set computed in the first phase and splits its elements into the array of sets N_A according to the terminating node of the arcs. The algorithm also modifies the computed sets in order to detect nodes that can be merged and to define packs and put them into the queue Q. The time complexity of this algorithm depends only on the number of neighbors of a node, because thanks to the lists we never reach an empty array. In addition each arc is traversed only twice: one for the value and one for the node. The space complexity is in $O(n+d)$.

The reduction of the whole MDD is made by applying a BFS from the bottom to the top and by calling Function REDUCELAYER for each layer with L the list of nodes to merge as parameter.

5 A Generic Apply Function

In a way similar as Bryant in his paper about BDDs we propose to define efficient algorithms for combining MDDs. An algorithm for computing the intersection of two MDD has been proposed in [Bergman *et al.*, 2014].

We present a general algorithm defining several operators. From the MDDs mdd_1 and mdd_2 it computes $mdd_r = mdd_1 \oplus mdd_2$, where $\oplus$ is union, intersection, difference, symmetric difference, complementary of union and complementary of intersection. In addition, we give an efficient algorithm for computing the complementary of an MDD.

Our algorithm is generic and mainly based on the possible combinations of labeled arcs. It proceeds by associating nodes of the two MDDs. Each node x of the resulting MDD is associated with a node x_1 of the first MDD and a node x_2 of the second MDD. So, each node of the resulting MDD can be represented either by an index, or by a pair (x_1, x_2). First, the root is created from the two roots. Then, we build successively the layers. From the nodes of layer $i-1$ we build the nodes of layer i as follows. For each node $x = (x_1, x_2)$ of layer $i-1$, we consider the arcs outgoing from nodes x_1 and x_2 and labeled by the same value v. We recall that there is only one arc leaving a node x with a given label. Thus, there are four possibilities depending on whether there are y_1 and y_2 such that (x_1, v, y_1) and (x_2, v, y_2) exist or not. The action that we perform for each of these possibilities will define the operation we perform for the given layer. For instance, a union is defined by creating a node $y = (y_1, y_2)$ and an arc (x, v, y) each time one of the arcs (x_1, v, y_1) or (x_2, v, y_2) exists. An intersection is defined by creating a node $y = (y_1, y_2)$ and an arc (x, v, y) when both arcs (x_1, v, y_1) and (x_2, v, y_2) exist. Thus, these operations can be simply defined by expressing the condition for creating a node and an arc.

Algorithm 2 Complementary Operation.

```
COMPLEMENTARY(L, mdd_1, V)
  // L[i] is the set of nodes in layer i.
  root ← CREATENODE(root(mdd_1))
  L[0] ← {root}
  for each i ∈ 1..r − 1 do
      L[i] ← ∅
      for each node x ∈ L[i − 1] do
          get x_1 from x = (x_1, nil)
          for each v ∈ V[i] do
              if ∃(x_1, v, y_1) ∈ ω^+(x_1) then
                  ADDARCANDNODE(L, i, x, v, y_1, nil)
              else
                  MANAGEWILDCARDPATH(i, w)
                  CREATEARC(L, i, x, v, w[i])
  L[r] ← t
  for each node x ∈ L[r − 1] do
      get x_1 from x = (x_1, nil)
      for each v ∈ V[r] do
          if ∄(x_1, v, y_1) ∈ ω^+(x_1) then
              CREATEARC(L, i, x, v, t)
  PREDUCE(L)
  return root
```

In order to define more operations we need to extend this idea in two ways. First, we need to be able to deal with the absence of both arcs and, then, we need to make a special treatment for the last layer. A good example for understanding these needs is the definition of the complementary of a set. In this case, we need to create an arc in mdd_r when there is no arc in mdd_1. Consider a node x_1. If there is no arc

labeled by v outgoing from x_1 then such an arc must be in mdd_r the complementary of mdd_1. How can we define this arc, because it has no terminating node? For answering this question we create a single particular node for each layer i: the wild-card node denoted by $w[i]$ and we use it when a node is needed. This means that when there is no arc (x_1, v, y_1) for the couple (x_1, v) then we create the arc $(x, v, w[i])$ in mdd_r. In addition we link together wild-card nodes of consecutive layers with all possible values by creating arcs of the form $(w[i], v, w[i+1])$ for each value v. Consider for instance the MDD defined from the tuples (a, a, a), (b, b, b) and (c, c, c). The complementary MDD contains the tuples $(a, \{b, c\}, *)$ which belong to the complement of (a, a, a). Thus, we have a link from x_a, the node reached from the root by the arc labeled a, to the wild-card node $w[2]$ with b and c as label. However, tuple $(a, a, \{b, c\})$ is also in mdd_r. Thus, we also need to represent the arc (x_1, a, y_1) of mdd_1 in mdd_r. Therefore, if there is an arc in mdd_1, then mdd_r must also contain the same arc. However, this cannot be true for the last layer; otherwise mdd_1 would be included in mdd_r. For this layer we should not represent an arc belonging to the initial MDD. Therefore, specific rules of arc creations must be applied for the last layer. Algorithm 2 is a possible implementation of this operation.

The generic algorithm we propose is defined by the application or not of a procedure adding a node and creating an arc for the four possible cases of arc existence. Function APPLY, given in Algorithm 3 takes as parameters the two MDDs and two arrays op and V having as many elements as layers. For each layer i, $op[i]$ contains 4 entries, each one representing the fact that we create an arc or not for a combination of arc existence in the two MDDs and $V[i]$ represents the set of values needed by the complementary set. If it is equal to nil then $V[i]$ will be equal to the union of the values of the neighbors of the considered nodes. The values of $op[i]$ defining the binary operations are defined as follows for the different combinations:

	op[0] $\neg a1 \wedge \neg a2$		op[1] $\neg a1 \wedge a2$		op[2] $a1 \wedge \neg a2$		op[3] $a1 \wedge a2$	
layer	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r	[1..r-1]	r
$A \cap B$	F	F	F	F	F	F	T	T
$A \cup B$	F	F	T	T	T	T	T	T
$A - B$	F	F	F	F	T	T	T	F
$A \Delta B$	F	F	T	T	T	T	T	F
$\overline{A \cup B}$	T	T	T	F	T	F	T	F
$\overline{A \cap B}$	T	T	T	T	T	T	T	F

Function MANAGEWILDCARDPATH creates a wild-card node and the arcs between wild-card nodes when at least one wild-card node is needed in mdd_r.

6 Experiments

The experiments have been made on a 6 cores server (Intel 3930) having 64GB of memory and running under Windows 7. The algorithms have been implemented on the top of or-tools solver from Google [Perron, 2013] version 3158.

Motivating Example. This work has been mainly motivated by some phrase generation problems and notably the one defined in [Papadopoulos *et al.*, 2014]. This problem deals with Markov Sequence Generation on corpus having more than 10,000 words. The goal is to generate phrases having 24 words where all successions of 4 words come from the corpus and where there is no sequence of more than 8 words coming from the corpus. By replacing the corpus by sequences of 4 words of the corpus and by linking them together when two sequences have 3 words in common, we define the contracted corpus and we can reduce the size of the problem because we can only consider forbidden sequences of $8 - 4 = 4$ words. We propose to model this problem by MDDs expressing sequences of words. Values of variables are words of the corpus, so we have a huge number of values. From an initial MDD representing allowed sequence of 4 words we perform some intersections of MDD until obtaining an MDD of size 20.

More precisely, first we define mdd_4 the MDD containing all the sequences of 4 words from the contracted corpus, that is sequence of 8 words in the initial corpus. Then, we define an MDD having 4 variables from all the sequences of 2 words

Algorithm 3 Generic Apply Function.

```
APPLY(mdd_1, mdd_2, op, V): MDD
  // L[i] is the set of nodes in layer i.
  root ← CREATENODE(root(mdd_1), root(mdd_2))
  L[0] ← {root}
  for each i ∈ 1..r do
      L[i] ← ∅
      for each node x ∈ L[i − 1] do
          get x_1 and x_2 from x = (x_1, x_2)
          if V[i] = nil then
              V[i] ← VALUES(ω^+(x_1) ∪ ω^+(x_2))
          for each v ∈ V[i] do
              if ∄(x_1, v, y_1) ∈ ω^+(x_1) then
                  if ∄(x_2, v, y_2) ∈ ω^+(x_2) ∧ op[0] then
                      MANAGEWILDCARDPATH(i, w)
                      CREATEARC(L, i, x, v, w[i])
                  if ∃(x_2, v, y_2) ∈ ω^+(x_2) ∧ op[1] then
                      ADDARCANDNODE(L, i, x, v, nil, y_2)
              else
                  if ∄(x_2, v, y_2) ∈ ω^+(x_2) ∧ op[2] then
                      ADDARCANDNODE(L, i, x, v, y_1, nil)
                  if ∃(x_2, v, y_2) ∈ ω^+(x_2) ∧ op[3] then
                      ADDARCANDNODE(L, i, x, v, y_1, y_2)
  merge all nodes of L[r] into t
  PREDUCE(L)
  return root

ADDARCANDNODE(L, i, x, y_1, v, y_2)
  if ∄y ∈ L[i] s.t. y = (y_1, y_2) then
      y ← CREATENODE(y_1, y_2)
      add y to L[i]
  CREATEARC(L, i, x, v, y)
```

from the contracted corpus (Markov MDD) and we substract mdd_4 from it in order to obtain mdd_a the MDD containing allowed sequences of 4 words that can be made from the contracted corpus, and forbidding plagiarism sequences. Then, we repeatedly define mdd_a for each sequence of 4 variables in the ordered set: $x_1, ...x_{20}$. That is we define 16 MDDs. Next, we successively intersect the MDDs. This means that we intersect the MDD defined on $x_1, .., x_i$ with the MDD defined on $x_{i-2}, ..., x_{i+1}$ for obtaining the MDD defined on $x_1, .., x_{i+1}$. For intersecting a pair of MDDs defined on different variables we modify them by adding variables accepting all the possibles values. More precisely, the MDD defined on $x_1, .., x_i$ is transformed into the MDD defined on $x_1, .., x_{i+1}$ where each values of x_{i+1} is compatible with any path of the first MDD. This corresponds to a duplication of the last layer. Similarly, we will duplicate several times the first layer of the MDD defined on $x_{i-2}, ..., x_{i+1}$ to add variables from x_1 to x_i into it. After these operations, mdd_r is the final MDD corresponding exactly to the automaton of the dedicated method defined in [Papadopoulos *et al.*, 2014]. Note that all MDDs are reduced.

The main issue with this approach is the size of the MDDs. The contracted corpus has 10,785 words. Markov MDD has 15,950 nodes and 129,465 arcs, mdd_4 has 56,225 nodes and 127,786 arcs, mdd_r has 1,208,219 nodes and 188,035,203 arcs. The reduction is efficient, for instance for mdd_4 the number of nodes go from 123,025 to 56,225 and mdd_r has 2.2 times fewer node than the automaton.

With the new algorithms we propose it needs 425s to build mdd_r, and finding the 50 first solutions takes 26.8s. The whole process requires 7min 31s. The building time is slightly more (about 20%) than the dedicated algorithm given in [Papadopoulos *et al.*, 2014] but the solving time is similar [Papadopoulos,].

Algorithms like `mddc` of Cheng and Yap cannot be used for making the intersections and applying the reduction operations, because the memory consumption exceeded quickly 64GB whereas it is kept below 10 GB with our algorithms. Thus, we tried a different approach based only on the set of MDDs for each sequence without intersecting them. For a set of 9 variables (instead of 20) it takes more than 50 min with `mddc` to find the 50 first solutions whereas it took 6 seconds for MDD4R with mdd_r. It is a gain factor of 500.

Now, we propose to detail some different improvements obtained by our algorithms.

Comparison of Reduce Functions. We select some problems from the Solver Competition archive [Lecoutre, 2009]. For each type of problem we compute the geometric mean of the reduction times of all the instances for the pReduce algorithm and `mddify` the algorithm of Cheng and Yap. We obtain the following results which clearly show the advantage of our method.

type of problem	pReduce	`mddify`
rand-5-12-12-200-p12442	8.2	46.7
rand-8-20-5-18-800	74.5	191.8
crossword-m1c-uk-vg	50.2	668.2
crossword-m1c-ogd-vg	103.5	724.6
crossword-m1c-lex-vg	5.0	97.0
bdd-21-133-18-78	110.9	244.0

We also compare the two methods on random table constraints. The following tables show the gain factor of our approach.

	1000 tuples			
	arity			
d	6	8	10	12
12	11	20.6	26.7	29.8
30	32.3	47.5	57.8	54.7
60	80.6	84.6	76.1	79.1

	arity = 12		
	tuples		
d	1K	10K	100K
4	8.2	5.3	5.9
8	20	18.7	18.6
12	29.8	25,8	38.6
30	54.7	40.1	110.6
60	79.1	55.1	150.0

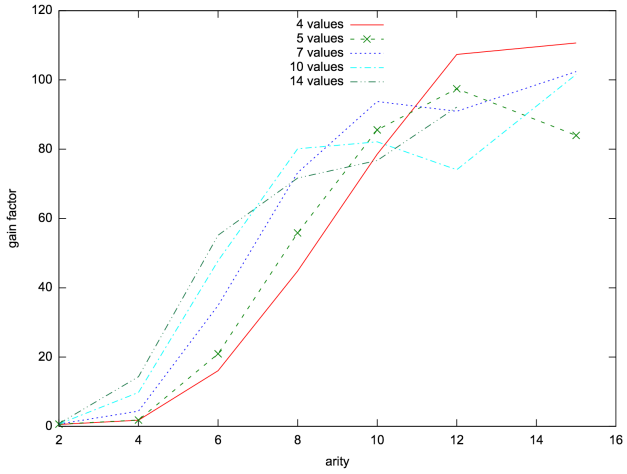
Comparison of Creations from Regular Constraints We use all the regular constraints defined from the pentominoes-int problems of the 2014 Minizinc Challenge, because an efficient algorithm is required to solve them. We compare the creation from a regular constraint that we propose versus the creation of the graph (which is a kind of MDD) performed by Pesant. The following table show the gain factors we obtain:

	min	max	average
gain factor	2.0	5.3	4.1

Ternary decomposition vs MDD + MDD4R

It is often considered (See [Beldiceanu *et al.*, 2004; Quimper and Walsh, 2006a; 2006b]) that the best way for maintaining arc consistency for regular constraint is to decompose the constraint into a set of ternary transition constraints and to directly deal with them. We propose to compare this model with the explicit use of the MDD corresponding to the automaton of the regular constraint in conjunction with MDD4R algorithm. The MDD is reduced.

We use constraints defined by transition constraints involving 8,000 tuples. The following figure gives the factor of gains of the use of a MDD + MDDR4 in comparison with transition constraints + GAC4R and clearly shows the advantage of our approach.



We also compare the two approaches on a problem with 5 random constraints and one knapsack constraint imposing that the sum of all variables must be greater than a value k (usually defined as the mean of the domains). The results given in the following table should a gain factor of 1.4:

Arity	dom size	1 sol		all sol	
		Ternary	MDD4R	Ternary	MDD4R
8	6	0.7	0.3	18.4	12.2
8	8	0.8	0.5	25.1	17.2
10	8	0.9	0.4	44.3	31.8
10	10	1.3	0.7	58.4	41.3
12	10	2.3	1.5	89.2	66.4
12	12	4.2	2.8	109.6	82.5

7 Conclusion

We have proposed new efficient algorithms for creating and reducing Multi-Valued Decision Diagrams (MDDs). The new reduction algorithm has an $O(n+m+d)$ space and time complexity and so may be used for huge MDDs. We have also introduced a generic apply function from which we can define the most common operations on MDDs: intersection, union, difference, symmetric difference, complement of union and complement of intersection. We experiment our approach against the previous ones and on a complex phrase generation problem and we show that a model using our algorithms and MDD4R is competitive with dedicated algorithms defining complex automata. Some other experiments demonstrate the improvements brought by our algorithms.

References

[Andersen *et al.*, 2007] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *CP*, pages 118–132, 2007.

[Beldiceanu *et al.*, 2004] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *CP'04*, pages 107–122, 2004.

[Bergman *et al.*, 2011] David Bergman, Willem Jan van Hoeve, and John N. Hooker. Manipulating mdd relaxations for combinatorial optimization. In *CPAIOR*, pages 20–35, 2011.

[Bergman *et al.*, 2014] D. Bergman, A. Cire, and W-J. van Hoeve. Mdd propagation for sequence constraints. *Journal of Artificial Intelligence Research*, 50:697–722, 2014.

[Bryant, 1986] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.

[Cheng and Yap, 2010] K. Cheng and R. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15, 2010.

[Gange *et al.*, 2011] G. Gange, P. Stuckey, and Radoslaw Szymanek. Mdd propagators with explanation. *Constraints*, 16:407–429, 2011.

[Hadzic *et al.*, 2008] Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *CP*, pages 448–462, 2008.

[Hoda *et al.*, 2010] Samid Hoda, Willem Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *CP*, pages 266–280, 2010.

[Lecoutre, 2009] Christophe Lecoutre. Csp/maxcsp/wcsp solver competitions. In *http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html*, 2009.

[Papadopoulos,] A. Papadopoulos. Personnal communication.

[Papadopoulos *et al.*, 2014] A. Papadopoulos, P. Roy, and F. Pachet. Avoiding plagiarism in markov sequence generation. In *Proceeding of the Twenty-Eight AAAI Conference on Artificial Intelligence*, pages 2731–2737, 2014.

[Perez and Régin, 2014] G. Perez and J-C. Régin. Improving GAC-4 for table and MDD constraints. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 606–621, 2014.

[Perron, 2013] L. Perron. Or-tools. In *Workshop "CP Solvers: Modeling, Applications, Integration, and Standardization"*, 2013.

[Pesant, 2004] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proc. CP'04*, pages 482–495, 2004.

[Quimper and Walsh, 2006a] C-G. Quimper and T. Walsh. Global grammar constraints. In *CP'06*, pages 751–755, 2006.

[Quimper and Walsh, 2006b] C-G. Quimper and T. Walsh. Global grammar constraints. Technical report, Waterloo University, 2006.

[Trick, 2003] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research*, 118:73 – 84, 2003.